Advanced Python Exercises

Exercise 1: Working with Data

For this exercise, we're going to load GPS data in as points and then create line segments with estimated speed between the points.

1) Look in the 'gpsTrack' directory. Note that there is a geodatabase (empty) and a GPX file. We can import the GPX file as points using built-in tools, but to generate lines, we'll need to read those points, calculate speed, and write out to a new feature class.
2) Open IDLE and create a new Python file in the 'gpsTrack' directory. Import arcpy.
3) Let's set up some structure in this script using functions. We use the def statement to create a function, along with its name and any parameters it receives. Create a function called main by typing in the following:
```
def main(inGPSfile, outGPSline):
```
4) Let's set up a basic default. We'll need several times through the script the coordinate system we want to store the information in. To set that, we need to create a SpatialReference object. In this case, I want to use MD State Plane (with measurement unit of feet), which is appropriate for the area where the GPS was recorded; it has a wkid of 2248. Type the following in the script under and indented in the def main line:
```
statePlane = arcpy.SpatialReference(2248)
```
5) Now we can load the GPS points into a feature class. As I mentioned at the beginning, there's a built-in script for that. We'll store the output in the in_memory workspace, a workspace created by ArcGIS which is both temporary and quick to access, which are both benefits when working with scripts:
```
#Load the points
arcpy.AddMessage('Reading Points')
inPoints = arcpy.GPXtoFeatures_conversion(inGPSfile, 'in_memory\\gpsPoints')
```
6) Now that we've loaded the points into a format ArcGIS can work with, we need to make them accessible so that we can calculate speed between points. To do that, we can load the points into Python's memory using a SearchCursor. We'll create an empty list to store the read results. We only need the X,Y, and timestamp information; as part of setting up the SearchCursor, we can limit the data we get back to just that info. Finally, we need to load the coordinate information in the state plane coordinate system we previously set up (note that the last 2 lines printed below are 1 line in the script):
```
points = []
with arcpy.da.SearchCursor(inPoints, ['SHAPE@X', 'SHAPE@Y', 'dateTime'],
spatial_reference=statePlane) as rows:
```
7) Now that we've opened up the points, we'll add them to the array to access them. The following will be indented inside the **with** statement:
```
for row in rows:
    points.append(row)
```
8) With the points in hand, we can now calculate the speed and create the line segments. To do so, we again need to set up some variables to store the results of the calculations and keep track of where we are in the calculations:
```
lines = []
p = 1
arcpy.AddMessage('Calculating speed')
```

9) We can use a while loop to calculate the point-by-point distance. To calculate distance, we need to use the math.sqrt function, so add the math module to the import statement at the top. Then we can get the time elapsed from the time stamp and then calculate the speed:

```
    while p < len(points):
        thisPoint = points[p]
        lastPoint = points[p-1]
        #calculate distance
        distance = math.sqrt((thisPoint[0] - lastPoint[0])**2 + (thisPoint[1] -
lastPoint[1])**2)
        #calculate time difference
        timeTaken = thisPoint[2] - lastPoint[2]
        #calculate speed (ft/sec because of the coordinate system
        speedFPS = distance / timeTaken.seconds
        #Convert to MPH 3600 sec/hr & 1 mi / 5280 ft
        speedMPH = speedFPS * 3600.0 / 5280.0
```

10) Then to create the line, we use the point coordinates to create the points and then create the line. Then we add the line & speed to list of lines and move on to the next point. The following uses a technique called list comprehension to create the points- for each item in the pointCoords list (there 2, each with X & Y), we create a point with the information. Again, this still should be indented inside the while loop:

```
        #Create the line from the 2 base points
        pointCoords = [[lastPoint[0], lastPoint[1]], [thisPoint[0], thisPoint[1]]]
        linePoints = [arcpy.Point(*coords) for coords in pointCoords]
        thisLine = arcpy.Polyline(arcpy.Array(linePoints), statePlane)
        #add the line to out list
        lines.append((thisLine, speedMPH))
        p = p+1
```

11) Now let's create the output line feature class and add a field to store the speed:

```
    #Create the output feature class
    arcpy.AddMessage('Creating Output')
    outFC = arcpy.CreateFeatureclass_management(arcpy.env.workspace, outGPSline,
"POLYLINE", spatial_reference=statePlane)
    arcpy.AddField_management(outFC, "SPEED_MPH", "FLOAT")
```

12) Just as we used a SearchCursor to read the feature information, we can use an InsertCursor to write the information, using the insertRow method:

```
    #Write in the lines
    arcpy.AddMessage('Writing Output')
    with arcpy.da.InsertCursor(outFC, ["SHAPE@", "SPEED_MPH"]) as insert:
        for line in lines:
            insert.insertRow(line)
```

13) Now that we have the main function done, we need to be able to accept the parameters when they come in. A way to do that is to have a set of commands set to run when this script is run from the command line or as a script tool. This is flagged when the private *__name__* variable is set to '*__main__*'. So let's test for that, and then get the parameters and run the function. The first line below is unindented, with the following lines indented under it:

```
if __name__ == '__main__':
    inFile = arcpy.GetParameterAsText(0)
    outFC = arcpy.GetParameterAsText(1)
    main(inFile, outFC)
```

14) Test out the script by either making a script tool out of it or running it on the command line. Another way to test it would be to replace the inFile & outFC assignments with hardcoded values. The script should look like this if done correctly:

```
import arcpy, math

def main(inGPSfile, outGPSline):
    statePlane = arcpy.SpatialReference(2248)

    #Load the points
    arcpy.AddMessage('Reading Points')
    inPoints = arcpy.GPXtoFeatures_conversion(inGPSfile, 'in_memory\gpsPoints')

    #Open a SearchCursor to read the points
    points = []
    with arcpy.da.SearchCursor(inPoints, ['SHAPE@X', 'SHAPE@Y', 'dateTime'], spatial_reference=statePlane) as rows:
        for row in rows:
            points.append(row)

    lines = []
    p = 1
    arcpy.AddMessage('Calculating speed')
    while p < len(points):
        thisPoint = points[p]
        lastPoint = points[p-1]
        #calculate distance
        distance = math.sqrt((thisPoint[0] - lastPoint[0])**2 + (thisPoint[1] - lastPoint[1])**2)
        #calculate time difference
        timeTaken = thisPoint[2] - lastPoint[2]
        #calculate speed (miles per hour)
        speedFPS = distance / timeTaken.seconds
        speedMPH = speedFPS * 3600.0 / 5280.0
        #Create the line from the 2 base points
        pointCoords = [[lastPoint[0], lastPoint[1]], [thisPoint[0], thisPoint[1]]]
        linePoints = [arcpy.Point(*coords) for coords in pointCoords]
        thisLine = arcpy.Polyline(arcpy.Array(linePoints), statePlane)
        #add the line to out list
        lines.append((thisLine, speedMPH))
        p = p+1

    #Create the output feature class
    arcpy.AddMessage('Creating Output')
    outFC = arcpy.CreateFeatureclass_management(arcpy.env.workspace, outGPSline, "POLYLINE", spatial_reference=statePlane)
    arcpy.AddField_management(outFC, "SPEED_MPH", "FLOAT")

    #Write in the lines
    arcpy.AddMessage('Writing Output')
    with arcpy.da.InsertCursor(outFC, ["SHAPE@", "SPEED_MPH"]) as insert:
        for line in lines:
            insert.insertRow(line)

if __name__ == '__main__':
    inFile = arcpy.GetParameterAsText(0)
    outFC = arcpy.GetParameterAsText(1)
    main(inFile, outFC)
```

Exercise 2: Working with Map Elements & Data-Driven Pages

This exercise will show you how to manipulate elements in the map layout and generate print output.

1) Open up MapProduction\WetlandAtlas.mxd. This is a map document designed to produce a simple atlas of wetlands by county using ArcMap's Data Driven Pages functionality. To access the Data Driven Pages features, right-click in the toolbar area and open the 'Data Driven Pages' toolbar, which will allow you to navigate from county to county. Also note the text box to list adjacent counties – open up its properties; you should see that it has the 'Element Name' of 'adjacent'. We'll use that to fill it in for each county.

2) Create a new script in IDLE/PyScripter and **import arcpy**. Also import the **os** library, which manipulates file names. For this script, we're going to create 2 functions, in addition to the parts that run automatically (inside the if __name__ = '__main__': statement):
    a. A function to generate the maps
    b. A function to add in the adjacent county list
3) Let's start by simply creating the functions in the script. Enter in the following:

```
#Generate the maps
def generateMaps(mxd, outDirectory, nameField):
    pass

#Find the adjacent counties and list them
def findAdjacentCounties(mxd, nameField):
    pass
```

The pass command tells Python to exit the statement; it's useful as a placeholder as we develop the script. The findAdjacentCounties function will be called within the generateMaps function; it needs to be run for each map.

4) Let's also put in the information we need when the script runs:

```
if __name__ == '__main__':
    mxdName == arcpy.GetParameterAsText(0)
    if mxdName == None or mxdName == "#":
        mxdName = <Get mxd path from instructor>
    mxd = arcpy.mapping.MapDocument(mxdName)
    outDir = arcpy.GetParameterAsText(1)
    nameField = arcpy.GetParameterAsText(2)
    generateMaps(mxd, outDir, adjacentLayer, nameField)
```

This gets the inputs in and also starts to use the arcpy.mapping module, giving us access to the map document.

5) We can fill in the code we need to generate the maps before getting the adjacent list. This will make use of the map document's *dataDrivenPages* object to set which map page we're printing at a time (refer to http://resources.arcgis.com/en/help/main/10.2/#/DataDrivenPages/00s300000030000000/). Place the following code within the generateMaps function:

```
mapPage = 1
while mapPage <= mxd.dataDrivenPages.pageCount:
    arcpy.AddMessage("Generating Page" + str(mapPage))
    mxd.dataDrivenPages.currentPageID = mapPage
    arcpy.AddMessage("Identifying counties")
    countylist = findAdjacentCounties (mxd, nameField)
    #We'll add more code here to process the counties once we finish that part
    arcpy.AddMessage("Generating PDF")
    arcpy.mapping.ExportToPDF(mxd, os.path.join(outDirectory, 'MD Wetlands ' +
str(mapPage) +".pdf"))
    mapPage = mapPage + 1
```

You can also delete the **pass** statement in the generateMaps function.

6) At this point, you can run the script (use 'NAME' for nameField and use the MapProduction folder for the out directory). You'll see a series of maps get generated.

7) Now let's work on finding the adjacent counties. We can get the currently selected county from the mapDocument.dataDrivenPages.pageRow object, which is the table entry for the currently selected feature. With the pageRow, we can then do the following:
    a. Get the county polygon of the selected item from the row
    b. Do a select by location with the polygon to get the counties that touch the polygon.
    c. Remove the original county from the selection, to just get the counties that touch.
    d. Copy the adjacent counties into a temporary feature class in memory and read out the names.
    e. Delete the temporary feature class to clean up for next time the function is called.
    f. Return the names from the function as a list.

8) To implement the above, we need to modify the findAdjacentCounties function. Remove the **pass** statement and add in the following to the function:

```
    #Get the polygon & name
    thisPoly = mxd.dataDrivenPages.pageRow.shape
    thisName = mxd.dataDrivenPages.pageRow.getValue(nameField)
```

This provides the current county in the atlas. The next part selects for the adjacent counties and removes the current county:

```
    #Select by location - we can use the dataDrivenPages index layer
    selectLayer = mxd.dataDrivenPages.indexLayer
    arcpy.SelectLayerByLocation_management(selectLayer, "INTERSECT", thisPoly)
    arcpy.SelectLayerByAttribute_management(selectLayer, "REMOVE_FROM_SELECTION",
'"' + nameField + '" = ' + "'" + thisName + "'")
```

9) The next part will copy the selection to a feature class we then read:

```
    #Copy the features to a temporary feature class
    tempFC = arcpy.CopyFeatures_management(selectLayer, 'in_memory\\tempSelection')
    #Read the names into a list
    countyNames = []
    #We can add a SQL clause to order the results
    with arcpy.da.SearchCursor(tempFC, [nameField], sql_clause=(None, "ORDER BY " +
nameField)) as rows:
        for row in rows:
            countyNames.append(row[0])
```

Finally, let's clean up the selection and temporary feature class and return the list:

```
    #Clean up the selection and temp feature class
    arcpy.DeleteFeatures_management(selectLayer, 'in_memory\\tempSelection')
    arcpy.SelectLayerByAttribute_management(selectLayer, 'CLEAR_SELECTION')
    #Return the list of county names
    return countyNames
```

10) Now that we're generating the list of counties, we need to process it in the generateMaps. After the comment line #We'll add more code here…., insert in a new line:

```
countyText = "Adjacent Counties\n"
```

This will start the piece of text in the map. Next we want to add the list of counties, each on their own line. In python, we can use the <string>.join() method to join the contents of a list with the specified string. To add the counties separated by newline characters, we would add:

```
countyText = countyText + "\n".join(countylist)
```

11) Now we need to place the text in the text element in the map. To do that, we need to search for the text element and access it's text property:

```
arcpy.mapping.ListLayoutElements(mxd, "TEXT_ELEMENT", "adjacent")[0].text =
countyText
```

Note that the ListLayoutElements function returns a list, even if only one element matches.

12) Try running the script. You'll see it works – mostly. Some of the counties in Maryland, like Prince George's County, have a single quotation mark (or apostrophe) in their name. When used in a Select by Attributes, we need to escape the quotation mark by placing a second single quotation mark along with it. To do that, insert the following lines immediately above the arcpy.SelectLayerByAttribute statement:

```
  #Escape any single quote marks
  thisName = thisName.replace("'", "''")
```

Now save the script, delete the pdf files already created and try re-running the script.

13) Again, for reference, a copy of the script is on the next page

```python
import arcpy, os

#Generate the maps
def generateMaps(mxd, outDirectory, nameField):
    mapPage = 1
    while mapPage <= mxd.dataDrivenPages.pageCount:
        arcpy.AddMessage("Generating Page" + str(mapPage))
        mxd.dataDrivenPages.currentPageID = mapPage
        arcpy.AddMessage("Identifying counties")
        countylist = findAdjacentCounties (mxd, nameField)
        #We'll add more code here to process the counties once we finish that part
        countyText = "Adjacent Counties\n"
        countyText = countyText + "\n".join(countylist)
        arcpy.mapping.ListLayoutElements(mxd, "TEXT_ELEMENT", "adjacent")[0].text = countyText
        arcpy.AddMessage("Generating PDF")
        arcpy.mapping.ExportToPDF(mxd, os.path.join(outDirectory, 'MD Wetlands ' + str(mapPage) +".pdf"))
        mapPage = mapPage + 1


#Find the adjacent counties and list them
def findAdjacentCounties(mxd, nameField):
    #Get the polygon & name
    thisPoly = mxd.dataDrivenPages.pageRow.shape
    thisName = mxd.dataDrivenPages.pageRow.getValue(nameField)
    #Select by location - we can use the dataDrivenPages index layer
    selectLayer = mxd.dataDrivenPages.indexLayer
    arcpy.SelectLayerByLocation_management(selectLayer, "INTERSECT", thisPoly)
    #Escape any single quote marks
    thisName = thisName.replace("'", "''")
    arcpy.SelectLayerByAttribute_management(selectLayer, "REMOVE_FROM_SELECTION", '"' + nameField + '" = ' + "'" + thisName + "'")
    #Copy the features to a temporary feature class
    tempFC = arcpy.CopyFeatures_management(selectLayer, 'in_memory\\tempSelection')
    #Read the names into a list
    countyNames = []
    #We can add a SQL clause to order the results
    with arcpy.da.SearchCursor(tempFC, [nameField], sql_clause=(None, "ORDER BY " + nameField)) as rows:
        for row in rows:
            countyNames.append(row[0])
    #Clean up the selection and temp feature class
    arcpy.DeleteFeatures_management('in_memory\\tempSelection')
    arcpy.SelectLayerByAttribute_management(selectLayer, 'CLEAR_SELECTION')
    #Return the list of county names
    return countyNames

if __name__ == '__main__':
    mxdName == arcpy.GetParameterAsText(0)
    if mxdName == None or mxdName == "#":
        mxdName = r'D:\Demos\FED\1406_FWS\python\advanced\mapProduction\WetlandAtlas.mxd'
    mxd = arcpy.mapping.MapDocument(mxdName)
    outDir = arcpy.GetParameterAsText(1)
    nameField = arcpy.GetParameterAsText(2)
    generateMaps(mxd, outDir, nameField)
```